الغاية من الجلسة:

برمجة كل من الخوارزميات التالية *UCS, Greedy,A

خوارزمية UCS:

تعتبر هذه الخوارزمية من خوارزميات البحث العمياء ولكنها وعلى عكس خوارزمية BFS تعطي حل أمثلي في حال كانت الكلف غير موحدة.

آلية عمل الخوارزمية:

خوارزمية ucs تستخدم رتل الأولوية (العقدة ذات الكلفة الأقل تكون ببداية الرتل).

ملاحظات:

- سمي رتل الأولوية بذلك لأن الكلفة الأقل توضع ببداية الرتل، بعكس الرتل المستخدم في BFS حيث أنه يضع الأبناء بترتيب عشوائي بغض النظر عن الكلفة الأقل
- تُحسب الكلفة دوما من العقدة الجذر.

هل الخوارزمية تامة؟(أي هل تجد حلاً بشكل دائم)

نعم بشرط ألا تكون الكلف صفرية وهو شرط نادر الحدوث أيضا. إذا كانت الكلف صفرية فمن الممكن أن يدخل في حلقة .

هل الخوارزمية optimal:

نعم بشرط ألا تكون هناك كلف سالبة.

خوارزمية UCS:

هي نفسها خوارزمية BFS ولكن الفرق الوحيد أن بنية المعطيات هي رتل أولوية .

```
package core.algorithms;
import core.*;
import java.util.*;
public class UCS extends SearchAlgorithm {
private PriorityQueue<Node> container = new
PriorityQueue<>();
private List<State> generated = new ArrayList<>();
private Collection<State> nextStates = new ArrayList<>();
private int visitingOrder = 0;
public UCS() {}
public UCS(boolean useGraphSearch) {
this.useGraphSeach = useGraphSearch;}
@Override
```

```
public Node search(Problem p) {
Node startNode = new Node(p.getInitialState())
container.add(startNode.getState());
initGraph(startNode);
if (p.isGoal(startNode.getState())) {
return startNode;}
container.add(startNode);
processes.add(new CollectionProccess("Push", startNode,
0));
while (!(container.isEmpty())) {

Node currentNode = container.poll();
processes.add(new CollectionProccess("Poll", currentNode,
0)); currentNode.setVisitingOrder(++visitingOrder);
if (useGraphSeach &&
explored.contains(currentNode.getState)) {continue;}
if (p.isGoal(currentNode.getState())) {
return currentNode;}
for (IAction action : p.getActions()) {
nextStates = action.apply(currentNode.getState());
developedNodes += nextStates.size();
for (State s : nextStates){
generated.add(s);
double actionCost = p.getActionCost(currentNode.getState(),
action, s);
double pathCost = actionCost + currentNode.getPathCost();
Node child = new Node(s, currentNode, action, pathCost);
container.add(child);
processes.add(new CollectionProccess("Push", child,
container.size() - 1));}}}
addNodeEdge(currentNode, child, action);
return null;

}

}
```

خوارزمية greedy:

Greedy /BestFirstSearc و*A

هذه الخوارزميات هي تقريبا نفس خوارزمية UCS الفرق الوحيد هو أن UCS تعتبر أن الكلفة الكلية هي كلفة الطريق الحقيقية totalCost=pathCost و Greedy تعتبر أن الكلفة الكلية هي قيمة التابع الحدسي totalCost=totalEstimatedCost

و* A تعتبر ن الكلفة الكلية هي مجموع لفة الطريق و قيمة التابع الحدسي totalCost=pathCost+totalEstimatedCost

لحساب قيمة التابع الحدسي نستدعي التابع الحدسي للمسألة p.getHeuristicFunction ونمرر له الحالة الحالية عبر getH(s) ي p.getHeuristicFunction().getH(s) فيصبح الكود لخوارزمية Greedy بالشكل الآتي:

```
package core.algorithms;
import core.*;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import java.util.PriorityQueue;
public class GreedyBestFisrtSeach extends SearchAlgorithm
{
private PriorityQueue<Node> container = new
PriorityQueue<>();
private List<State> explored = new ArrayList<>();
private Collection<State> nextStates = new ArrayList<>();
private int visitingOrder = 0;
public GreedyBestFisrtSeach() {


    public GreedyBestFisrtSeach(boolean useGraohSearch)
{
this.useGraphSeach = useGraohSearch;}
@Override
public Node search(Problem p) {
Node startNode = new Node(p.getInitialState());
container.add(startNode);
proccesses.add(new CollectionProccess("Push", startNode));
initGraph(startNode);
while (!(container.isEmpty())) {
Node currentNode = container.poll();
proccesses.add(new CollectionProccess("Poll", currentNode));
```

```
currentNode.setVisitingOrder(++visitingOrder);
                                                    if (useGraphSeach &&
                                explored.contains(currentNode.getState())) {

continue;}
if (p.isGoal(currentNode.getState())) {
return currentNode;}
explored.add(currentNode.getState());
                                        for (IAction action : p.getActions()) {

nextStates = action.apply(currentNode.getState());
developedNodes+=nextStates.size();
for (State s : nextStates) {
double actionCost = p.getActionCost(currentNode.getState(),
action, s);
double pathCost = actionCost + currentNode.getPathCost();
double totalEstimatedCost =
p.getHeuristicFunction().getH(s);
Node child = new Node(s, currentNode, action, pathCost,
totalEstimatedCost);
container.add(child);
proccesses.add(new CollectionProccess("Push", child));
addNodeEdge(currentNode, child, action);}
return null


}}
```

```
package core.algorithms;
import core.*;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import java.util.PriorityQueue;
import scala.None;
public class Astar extends SearchAlgorithm {
private PriorityQueue<Node> container = new
PriorityQueue<>();
                private List<State> explored = new ArrayList<>();
private Collection<State> nextStates = new ArrayList<>();
private int visitingOrder = 0;
public Astar() {}
public Astar(boolean useGraphSearch) {
this.useGraphSeach = useGraphSearch;}
@Override
public Node search(Problem p) {
                    Node startNode = new Node(p.getInitialState());
container.add(startNode);
proccesses.add(new CollectionProccess("Push", startNode));
initGraph(startNode);
while(!(container.isEmpty())) {
Node currentNode = container.poll();
proccesses.add(new CollectionProccess("Poll", currentNode));
currentNode.setVisitingOrder(++visitingOrder);
if(useGraphSeach &&
explored.contains(currentNode.getState())) {
continue;}
if (p.isGoal(currentNode.getState())) {
return currentNode;}
explored.add(currentNode.getState());
for(IAction action : p.getActions()) {
nextStates = action.apply(currentNode.getState());
```

```
for(State s : nextStates) {
double actionCost = p.getActionCost(currentNode.getState(),
action, s);
    double pathCost = actionCost + currentNode.getPathCost();

double totalEstimatedCost =
p.getHeuristicFunction().getH(s)  + pathCost;
developedNodes++;
Node child = new Node(s, currentNode, action, pathCost,
totalEstimatedCost);
container.add(child);
proccesses.add(new CollectionProccess("Push", child));
addNodeEdge(currentNode, child, action);}
}}
return null;}}
```